VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Kontejnerizace, migrace a škálovatelnost databázových technologií

Containerization, Migration and Scalability of Database Technologies

Jiří Čech

Bakalářská práce

Vedoucí práce: Ing. Radoslav Fasuga, Ph.D.

Ostrava, 2021

Abstrakt

Práce se zabývá problematikou tvorby a používání kontejnerizovaných databázových systémů. Popisuje vývoj virtualizace, který vedl ke kontejnerizaci. Zaměřuje se na kontejnerizační nástroj Docker a jeho verzi kontejnerizačního orchestrátoru Docker Swarm. Práce se zaměřuje primárně na dva databázové systémy, MySQL a Elasticsearch, u kterých se věnuje popisu jejich fungování a druhu replikačních nástrojů, které slouží k dosazení jejich vysoké dostupnosti v kontejnerizovaném prostředí. Dále práce zkoumá dopady virtualizovaného prostředí na výkon databází a zároveň popisuje klady datové redundance, které kontejnerizace přináší. Obsahuje zátěžové testy, které se snaží odpovědět na obavy ze ztráty výkonu při virtualizaci databází. Výsledkem je ukázka teoretických poznatků při návrhu a implementaci kontejnerizované webové aplikace, společně s teorií jejího škálování a testu vysoké dostupnosti.

Klíčová slova

Kontejnerizace, Docker, Elasticsearch, MySQL, vysoká dostupnost, Docker Swarm, replikace

Abstract

The thesis tackles challenges stemming from creation and usage of containerized database systems. It describes the evolution of virtualization that led to containerization. It focuses on Docker as a tool for achieving containerization and Docker's own version of a container orchestrator, Docker Swarm. The thesis focuses mainly on two database systems, MySQL and Elasticsearch, describes how they work and delves into their replication tools which serve to achieve high availability in a containerized environment. It analyzes the impact on performance in a virtualized environment and also describes the pros of data redundancy that containerization brings to the table. Contains stress tests which try to answer the concerns regarding performance of virtualized databases. The result is an analysis and an implementation of a fully containerized web application, which uses the theoretical findings of the thesis, along with tests of its scalability and high availability.

Keywords

Containerization, Docker, Elasticsearch, MySQL, high availability, Docker Swarm, replication

Poděkování

Rád bych na tomto místě poděkoval vedoucímu mé práce, Ing. Radoslavu Fasugovi, Ph.D., za odbornou pomoc, poskytnutí prostředků pro realizaci práce a hlavně trpělivost.

Obsah

Seznam použitých zkratek a symbolů

PaaS	–	Platform as a Service
VM	–	Virtual Machine
CLI	–	Command Line Interface
CPU	–	Central Processing Unit
RAM	–	Random Access Memory
API	–	Application Programming Interface
REST	–	Representational state transfer
SQL	–	Structured Query Language
CRUD	–	Create, Read, Update and Delete
TCP	–	Transmission Control Protocol
JSON	–	JavaScript Object Notation
DNS	–	Domain Name System
XML	–	Extensible Markup Language
HA	–	High Availability

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato práce se zaměřuje na popis možností, postupů a úskalí při škálování a kontejnerizaci databázových technologií. Kontejnery jsou bezpochyby jednou z největších změn v serverovnách a datacentrech za posledních pár let. Možnost roztříštit monolitické systémy na menší díly, které je možné automaticky násobit nebo ubírat podle aktuální potřeby, je bezpochyby výhodou pro mnoho aplikací. Jsou ale databáze jednou z těch aplikací?

V první kapitole je popsán základní kámen pro vznik kontejnerů, a to virtualizace. Popisuje její druhy a zkoumá jejich klady a zápory.

Další kapitola se již zabývá samotnou kontejnerizací a její architekturou. Popisuje technické pozadí kontejnerizačních nástrojů a zároveň i jak je běžně používat. Zaměřuje se hlavně na nástroj Docker, který je dnes synonymem pro kontejnery.

Třetí kapitola se ponořuje do problematiky celé práce. Analyzují se zde dopady výkonu fyzických i softwarových komponentů na fungovaní databází. Je zde sekce, která je zaměřena na popis mnoha druhů replikací dat mezi databázovými systémy. Jsou zde popisovány dva z hlavních databázových softwarů, které jsou navzájem technicky dost odlišné. Na konci kapitoly se práce věnuje už samotné problematice kontejnerizace. Zkoumá její klady a zápory, poukazuje na problémy specifické pro databáze v kontejnerech.

Následující část je věnována prezentaci výsledků ze zátěžového testování dvou databázových systémů v různých prostředích. Tato kapitola se snaží odpovědět na otázky z přechozích kapitol. Jak moc velký vliv má virtualizace na výkon databází? Pozná uživatel rozdíl mezi databází v kontejneru a mimo něj?

Poslední kapitola popisuje průběh vytvoření a nasazení kontejnerizované aplikace s vysoce dostupnou verzí databáze Elasticsearch. Je zde analýza systému, který bude kontejnerizován a následně s podrobným popisem implementován. Nechybí ani postup na přípravu prostředí pro kontejnery. Během celého procesu implementace je možné vidět poznatky z přechozích kapitol v praxi.

Kapitola 2

Virtualizace

Virtualizace je proces provozovaní virtuálních instancí operačních systémů na abstrahované vrstvě počítačového hardwaru. Umožnuje tedy provozovat naráz různé systémy na jednom fyzickém stroji. Pro aplikace běžící na virtualizovaném systému se vše jeví, jako kdyby byly spuštěny na klasickém dedikovaném stroji. [1]

2.1 Hypervisor

Hypervisor je software pro vytváření a provoz virtuálních strojů. Jedná se o vrstvu mezi hardwarem fyzického počítače a virtualizovaným systémem. Jeho hlavním úkolem je rozdělovaní zdrojů hosta (fyzický počítač) mezi virtuální stroje. Nejedná se pouze o výpočetní zdroje jako je CPU nebo RAM, ale jde také o poskytování uložiště, formou virtuálních HDD, nebo síťového připojení.

Hypervisory je možné rozdělit na dva typy podle toho, jak "blízko"je virtualizovaný systém k hardwaru hosta.

2.1.1 Hypervisor Typu 1

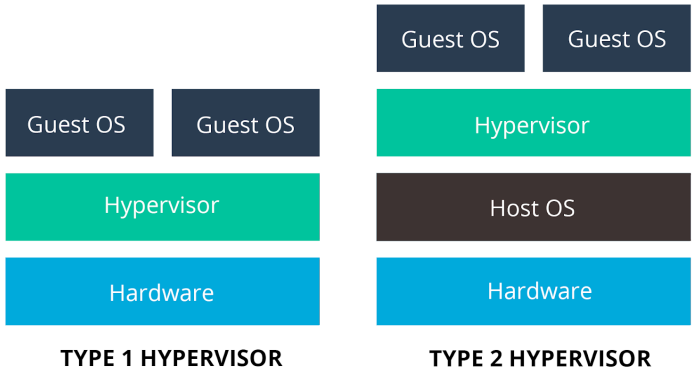
Hypervisor typu 1 je také nazýván jako *bare-metal* hypervisor. To znamená, že má přímý přistup k fyzickému hardwaru a nemusí spouštět žádný operační systém, pod kterým by běžel. Pro jeho funkčnost je ale potřeba, aby hardware tento přímý přístup podporoval. Jedná se o tzv. hardwarově akcelerovanou virtualizaci.
Tento přímý přístup mu poskytuje dvě hlavní výhody:

1. Rychlost

2. Bezpečnost

2.1.2 Hypervisor Typu 2

Na rozdíl od hypervisoru typu 1, typ 2 potřebuje pod sebou pro svůj chod operační systém. Nemůže tedy přímo komunikovat s fyzickým hardwarem a u různých systémových volání se musí spoléhat na daný operační systém. To má za následek to, že není tak výkonný jako typ 1, ale má větší kompatibilitu s různými hardwarovými konfiguracemi.



Obrázek 2.1: Typy hypervisorů [2]

Kapitola 3

Kontejnerizace

Jedná se o druh virtualizace na úrovni operačního systému. Na rozdíl od virtuálního stroje, který vytváří plnohodnotné prostředí operačního systému, si kontejnery toto prostředí "vypůjčí" od svého hosta a izolují pouze aplikační vrstvu. Aplikace běžící v kontejnerech sdílí přístup k jádru (kernel) operačního systému svého hosta. Díky tomu, že kontejnery sdílí stejný typ operačního systému jako host, je jejich provoz daleko více efektivní. Zároveň velikost obrazu (image) kontejneru může být velmi malý, i v řádech megabytů, jelikož stačí, aby tento obraz obsahoval pouze danou aplikaci. [3] Kontejnery jsou zabalené balíčky softwaru, které obsahují vše potřebné pro chod dané aplikace. Zmiňovaný kontejnerový obraz je snapshot stavu aplikace při vytvoření daného obrazu. Díky tomu je garantováno, že při každém spuštění bude kontejner ve stejném stavu. Daný obraz se během chodu kontejneru nijak nemění, kontejnery jsou tedy v základu bezstavové procesy.

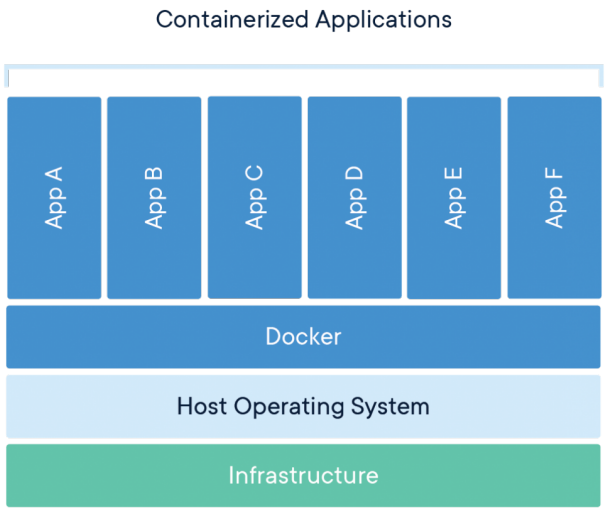
Schopnost zabalit aplikaci, která navíc bude výkonově velmi lehká, má za následek dvě z hlavních výhod kontejnerů v dnešní době. Přenositelnost a distribuce. Aplikace zabalená do kontejneru se dá jednoduše přenášet z počítače na počítač, a to navíc i s jistotou, že bude na obou z nich fungovat stejně. [4]

3.1 Nároky na operační systém

Kontejnery jsou aktuálně více dostupné na operačním systému Linux, ale existují i Windows kontejnery. Vzhledem k tomu ale jakým způsobem kontejnery sdílí operační systém hosta, není přenositelnost mezi těmito systémy nativně podporována. Například pro funkčnost Linux kontejneru na Windows stroji je potřeba podpůrného virtuálního Linux stroje, na kterém se následně kontejner spustí.

3.1.1 Linux

Kontejnery jsou daleko více používané v Linux prostředí, a to hlavně díky vlastnostem Linux jádra. Jedná se konkrétně o cgroups a namespaces. Tyto vlastnosti slouží hlavně pro izolaci procesů od



Obrázek 3.1: Architektura kontejnerizace [3]

ostatních.

Cgroups: Jde o vlastnost, která se stará o izolaci, limitaci a přidělování systémových prostředků pro proces nebo skupinu procesů. Je tedy možné například limitovat konkrétní proces na 20% CPU a 512MB RAM.

Namespaces: Umožnuje vytvořit jmenný prostor v systému, do kterého se následně mohou přidat různé procesy. Tento jmenný prostor obaluje globální systémové zdroje do abstrakce, která se pro procesy v rámci daného jmenného prostoru jeví jako jejich vlastní izolované zdroje. [5]

3.2 Docker

Docker je jedna z nejznámějších a nejpoužívanější implementací kontejnerových technologií. Firma Docker Inc. byla založena v roce 2010 a první verze byla vydána pod open-source licencí v březnu 2013. Jedná se sdružení několika PaaS nástrojů, které dohromady umožnují spravovat rozsáhlé kontejnerové aplikace. [3]

3.2.1 Architektura

Docker je postavený na client-server architektuře. Server komponent je tzv. daemon, což je proces, který běží na pozadí a v případě Dockeru spravuje vše od stavění, spouštění a distribuci Docker kontejnerů. Docker klient je komponent, který je nejblíže uživateli a umožnuje mu ovládat procesy

daemonu. Klient a server spolu komunikují přes REST API, a to buď přes UNIX socket, a nebo klasicky přes síť.

3.2.1.1 Daemon

Docker daemon (dockerd) naslouchá požadavkům poslaným na API a spravuje objekty Dockeru. Mezi tyto objekty patří obrazy (images), kontejnery, sítě a svazky (volumes). Daemon může také komunikovat s dalšími daemony kvůli správě služeb Dockeru.

3.2.1.2 Klient

Klient v Dockeru je aplikace, kterou uživatelé Dockeru komunikují s Dockerem daemonem. Při použití příkazů, jako je například "docker ps", klient přeloží tyto příkazy do správné formy pro REST API a odešle je daemonu, který je provede. Díky tomuto API nemusí být Docker klient pouze CLI aplikace, která je výchozí klientskou aplikací v Docker balíčku, ale může to být například i webová aplikace. Navíc klient může komunikovat s více než jedním daemonem, takže je možné takto spravovat všechny Docker servery z jednoho místa.

3.2.1.3 Obraz (Image)

Obraz v Docker terminologii označuje soubor, který slouží ke spuštění Docker kontejneru. Obraz je v podstatě uspořádaná kolekce kroků, nebo vrstev, které popisují běhové prostředí (runtime) kontejneru. Tyto kroky mohou být změny v konfiguraci aplikace, nebo nakopírovaní souboru z hostovského systému, ale převážně jde o instalaci aplikací či knihoven. Pro popis těchto kroků se používá Dockerfile. Ve výpisu kódu 3.1 je ukázka Dockerfile pro vytvoření upraveného kontejneru vizualizačního nástroje Kibana. Výsledný obraz bude mít 3 vrstvy. Příkaz *USER* nevytvoří novou vrstvu, takže vrstvy tvoří pouze příkaz *FROM*, *COPY* a *RUN*.

```
FROM docker.elastic.co/kibana/kibana:7.9.1

USER root

COPY --chown=1000:0 bin/ /usr/local/bin/
RUN chmod +x /usr/local/bin/startup.sh /usr/local/bin/kibana-docker

USER kibana
```

Listing 3.1: Ukázka Dockerfile

Vrstva je kolekcí změn v souborovém systému od stavu předešlé vrstvy. Tyto změny zahrnují přidaní, změnu nebo smazání souborů.

Může se zdát, že nároky na volný prostor na disku budou velké, když uvážíme, že Docker kontejner je běžící instancí Docker obrazu a těchto instancí z jednoho obrazu povětšinou běží několik desítek. Tento problém je vyřešen přidáním zapisovatelné vrstvy nad vrstvy obrazu pro každý běžící kontejner. Ostatní vrstvy obrazu zůstávají pouze pro čtení a mohou tak být sdíleny mezi více instancí bez nutnosti kopírovaní. Je-li potřeba upravit soubor ve vrstvě, která je pouze pro čtení, vytvoří se kopie tohoto souboru ve zmíněné zaposivatelné vrstvě a tato kopie je následně upravena. Zapisovatelná vrstva každého kontejneru je smazána po jeho vypnutí.

3.2.1.4 Registr

Docker registr je služba, která uschovává a poskytuje Docker obrazy přes Registry API. Nejznámějším Docker registrem je Docker Hub, který je možné procházet pomocí webového prohlížeče nebo Docker klientem. Docker Hub je výchozím registrem pro každou Docker instalaci. Je také možné si spustit vlastní soukromý registr.

3.2.1.5 Svazek (Volume)

Jak bylo zmíněno u Docker obrazu 3.2.1.3, je zapisovatelná vrstva pro každý kontejner pouze dočasná. To znamená, že ve výchozím stavu je každý kontejner bezstavový. Aby byla data kontejnerizované aplikace zachována i po restartu kontejneru, je nutné kontejneru přiřadit svazek. Docker umožnuje vytvořit svazek z velkého počtu poskytovatelů uložišť. Výchozí poskytovatel je přímo souborový systém hosta, na kterém Docker běží. Další poskytovatelé jsou např. NFS, CIFS, GlusterFS nebo VMware vSphere. Schopnost vytvoření svazku na síťovém poskytovateli uložiště, jako je například NFS, umožňuje spustit stejný kontejner na různých fyzických strojích (mají-li síťový přístup k NFS uložišti) a mít přístup ke stále stejným datům.

3.3 Orchestrace

S přibývajícím počtem kontejnerů roste i potřeba nějakým způsobem spravovat automaticky jejich vytvoření, nasazení a obecně jejich celý životní cyklus. Toto je hlavně důležité v dynamickém prostředí, kde běží stovky kontejnerů na různých fyzických strojích. V takovém prostředí se kontejnerový orchestrátor stává nutností. Kontejnerový orchestrátor spravuje mnoho různých procesů:

- Nasazení kontejnerů
- Redundance a dostupnost kontejnerů
- Znovu nasazení kontejneru při situaci, kdy je host nedostupný
- Správa zdrojů kontejnerů
- Škálování kontejnerů podle aktuální potřeby

- Monitorovaní zdraví kontejnerů
- Vyvažovaní zátěže mezi kontejnery
- Šítová komunikace mezi kontejnery
- Management uložišť pro data kontejnerů

Existuje mnoho různých orchestrátorů, které se od sebe liší počtem vlastností, a jednoduchostí nasazení a údržby. Mezi nejznámější patří:

- Docker Swarm
- Kubernetes
- OpenShift

3.3.1 Docker Swarm

Docker Swarm je software od Dockeru, který řeší orchestraci Docker kontejnerů. Princip fungovaní spočívá v propojení více Docker hostů za účelem rozprostření zátěže a redundance. Host může mít dvě role, manažer nebo pracovník.

Manažerský uzel je zodpovědný za správu celého Swarm clusteru. Vytváří služby, monitoruje stav služeb, poskytuje koncový bod pro Swarm HTTP API. Manažerské uzly mezi sebou používají Raft algoritmus pro udržení konzistentního stavu clusteru. [6]

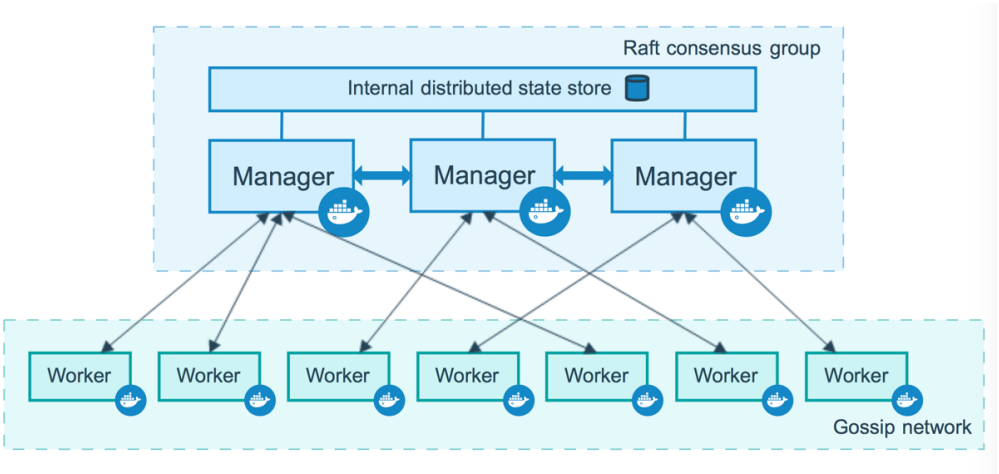
Pracovní uzly mají pouze jeden úkol, běh kontejnerů. Nejsou součástí konsensu pomocí Raft algoritmu, nedělají žádné rozhodnutí o tom, na jakém uzlu se má kontejner spustit a ani nemají API koncový bod. Manažerský uzel může být také zároveň i pracovníkem, takže je možné vytvořit Swarm cluster pouze s jedním uzlem. Pak ale cluster postrádá redundanci i možnost rozdělení zátěže, takže se jednouzlový Swarm používá pouze pro vývojové účely.

Základním stavebním kamenem Swarmu je *úloha (task)*. Úloha je ve své podstatě pouze kontejner běžící na uzlu specifikovaném manažerem. Manažer úlohy přiděluje na pracovní uzly podle definice služby.

Služba je definice jedné nebo více úloh. Služba může například být Wordpress stránka. Tato služba by se skládala z minimálně dvou úloh, webového serveru a databáze. Služby mohou fungovat ve dvou módech:

- Globální - služba běží na všech uzlech Swarm clusteru.
- Replikovaná - služba má specifikovaný počet replik, který má v danou chvíli ve Swarm clusteru běžet.

Další nastavení služeb je definice omezení (např. služba má běžet pouze na konkrétním uzlu), které mohou být specifikované značkami (label) nebo rolemi. Službám mohou být také přiřazeny přídavné zdroje, jako jsou svazky nebo sítě. Služby mohou také být sdruženy dohromady v konstruktu zvaném *stack*. Swarmu je možné přidělit úlohy pouze pomocí služeb nebo stacku. Na popis definice služby nebo stacku se používá *compose* soubor.



Obrázek 3.2: Docker Swarm architektura [6]

Kapitola 4

Databázový software

Databázový software je obecně software, který slouží pro persistentní uložení dat, a umožnuje čtení a zápis těchto dat.Tato práce se zaměřuje více na hardware požadavky databází než na jejich software. Software databáze nemusí nijak zajímat, že běží ve virtualizovaném prostředí tj. kontejneru. Interakce softwarových procedur s hardwarem je to, co může u kontejnerizované databáze ovlivnit výkon, efektivitu apod. Díky způsobům jakým virtualizační nástroje abstrahují hardwarové zdroje, je možné přidat zpoždění do procesů, které databáze vykonávají. Vzhledem k tomu, že tradičně jsou databázové aplikace konstruovány pro obsluhu co nejvíce dotazů za co nejkratší čas a zároveň zaručují konzistenci dat, jsou kladeny velké nároky na odezvu a výkon hardwaru. Víceméně všechny komponenty počítače se velkým dílem podílí na efektivitě databázového softwaru. [7]

- CPU - s rostoucím počtem transakcí rostou požadavky na výpočetní výkon. Databázové procesy jsou poměrně dost paralelizované (přístup k 10 různým datům najednou), takže počet výpočetních jader má velký vliv u databáze, která obsluhuje velké množství klientů.
- RAM - operační paměť slouží u databází převážně jako cache. Má-li databázový systém nedostatek RAM, musí častěji sahat pro data na disky, které jsou stále ještě pomalejší než RAM moduly. Dostatek výkonné operační paměti je dvojnásobně důležitý u tzv. *in-memory* databází (Redis [8], Memcached [9],. . .), které veškerá data drží pouze v paměti.
- Disk I/O - input/output (vstup/výstup) pevného disku. Počet I/O operací za určitý časový úsek je dost možná ten nejdůležitější parametr určující výkon u většiny databází. Aby byla zachována konzistence dat, databázový software musí počkat na potvrzení o zápisu dat na disk. SSD disky počty I/O operací několikanásobně navýšily oproti HDD, ale při virtualizaci uložiště je potřeba počítat s určitým snížením počtu I/O operací, jelikož mezi databázovým procesem a fyzickým zápisem na disk je nyní ještě virtualizační vrstva, která musí přeložit požadavek o zápis z virtuálního disku na disk fyzický. Toto snížení počtu I/O operací, resp. zvýšení zpoždění, může být nepostřehnutelné pro některé aplikace a pro jiné může být závažné [10] (např. burzovní systém).

- Síť - zpoždění síťového připojení může tvořit největší část celkového zpoždění od požadavku klienta po obdržení odpovědi na daný dotaz od databáze. Záleží, zda-li je klient v blízkosti databázového serveru, nebo je na druhé straně země. Toto zpoždění je ale převážně nemožné ovlivnit, jelikož narazíme na fyzikální limitace přenosu dat přes kabely. Virtualizační nástroje mohou přidat zpoždění dotazu kvůli virtualizaci síťového adaptéru, ale jde spíše o desetiny milisekundy, což může mít vliv pouze na ty časově nejcitlivější aplikace.

Tato práce se zaměří na dvě konkrétní databázové aplikace, MySQL a ElasticSearch. Obě tyto aplikace jsou dnes hojně využívány, ale přestavují dva různé pohledy na databáze.

4.1 Škálování

Škálování je proces, kdy se navýší počet aplikací nebo výpočetní výkon. Tyto dva způsoby se nazývají horizontální a vertikální škálování.

- **Horizontální** škálování přestavuje změnu počtu instancí aplikací či služeb. Mezi horizontální škálování se řadí i navýšení počtu samotných fyzických čí virtuálních serverů.

- **Vertikální** škálování znamená navýšení systémových prostředků. Přidání jádra procesoru do virtuálního stroje nebo navýšení operační paměti o 1GB jsou příklady tohoto škálování.

V dnešní době se může zdát, že vertikální škálování je jasnou volbou díky navyšování počtu výpočetních jader v jednom procesorovém čipu a navyšování velikosti adresovatelné paměti. Horizontální škálování má ale důležité výhody, které je nutné zvážit při rozhodování, který z druhů škálování využít. U horizontálního škálování je výhodou možnost nastavení vysoké dostupnosti a redundance. Vysoká dostupnost je princip, kdy je garantovaná funkčnost aplikace i při výpadku jednoho z jejich uzlů. Podle počtu uzlů dané aplikace je pak dán možný počet uzlů, které mohou vypadnout. Vysoká dostupnost zajišťuje, že se vyhneme tzv. *single point of failure*, což je situace, kdy by výpadek jednoho elementu znamenal výpadek celé aplikace.

Vertikální škálování je omezené dostupností hardwaru a také jeho cenou. Při navyšovaní požadavků na systém se firma může ocitnout v bodě, kdy už přidání nového hardwaru do serveru nepomůže se zahlcením systému. Horizontální škálování naopak poskytuje prostor pro optimalizaci ceny a rozvržení celé architektury systému. Je možné do produkce přidat více méně výkonných strojů, které jsou tak levnější a při dostatečném počtu i výkonnější než předchozí vertikálně škálované řešení. V dnešní době horizontální škálování představuje navyšovaní počtu virtuální serverů a kontejnerů.

U databázových systému je horizontální škálování realizovatelné různými metodami, master-slave nebo master-master replikace, anebo sharding.

4.1.1 Master-slave replikace

Master-slave replikace řeší hned více problémů. Problém s výkonem, zálohování nebo havárii databázového systému. Je běžná pro relační databáze.

Master-slave replikace umožňuje replikovat data z jednoho hlavního databázového serveru (master) na jeden nebo více dalších databázových serverů (slaves). Tyto slave servery se přes síť drží v synchronním stavu s master serverem pomocí replikačních protokolů. Slave servery slouží pouze jako koncový bod pro čtení, není možné na nich zapisovat do databáze. Master server slouží jak pro zápis, tak i čtení dat. Zvyšováním počtu slave serverů je tedy možné rozvrstvit čtecí operace a ulehčit tak databázovému systému.

Tento přístup sebou nese ale také nevýhody. Aplikace komunikující s databází musí být nastavena na odesílání požadavků o zápis na master server a požadavky na čtení rozvrstvit na slave servery, což zvyšuje komplexnost kódu a není plně flexibilní řešení. Při výpadku master serveru je aplikace v polofunkčním stavu, kdy může nadále číst data ze slave serverů, ale již nemůže zapisovat. Navíc proces povýšení slave serveru na master není automatický proces a může znamenat dočasnou nedostupnost aplikace nebo i ztrátu některých dat. Každý slave server navíc zvyšuje zátěž na master servery díky nutnosti kopírovat data na každý ze slave serverů.

4.1.2 Master-master replikace

Stejný princip jako u master-slave replikace, kdy se data synchronizují mezi více uzly. Na rozdíl od master-slave modelu je ale master-master replikace daleko více flexibilní a více redundantní. Každý uzel je totiž master a je tedy schopen jak čtení, tak zápisu do databáze. Při výpadku jednoho z uzlů nemusí tedy aplikace mít omezenou funkčnost, je-li vyřešeno automatické přepojení dotazu na funkční uzel.

Na rozdíl od master-slave replikace je master-master schopna dosáhnout vysoké dostupnosti. Všechny uzly jsou totožné a mají stejná data, takže nezáleží na tom, který z nich přestane fungovat.

4.1.3 Sharding

Sharding je metoda roztříštění velkého kusu dat na menší kousky (shards), kde každý shard může být na jiném stroji. Každý server funguje jako samostatná databáze a dohromady tvoří logický celek. Sharding umožnuje plynulé rozložení dotazů na databázové servery a tím i zajišťuje vysokou dostupnost. I při výpadku jednoho ze serverů není omezena funkčnost celkového datasetu. Sharding má velkou výhodu u obrovských datasetů, kdy díky jeho rozdělení na shardy je zachována vysoká propustnost dat. Další výhodou je ušetření místa na disku na jednotlivých uzlech, kdy dataset o velikosti 20TB může být rozdělen na 20 uzlů, kde každý shard zabere pouze 1TB.

4.2 MySQL

MySQL reprezentuje klasickou relační databázi, která byla vyvinuta za doby, kdy monolitické aplikace byly standardem a virtualizace nebyla tak rozšířená jako dnes. Je jednou z nejpoužívanějších relačních databází na světě. Dotazy nad daty zajišťuje jazyk SQL. V rámci MySQL je možné použít různé databázové enginy. Databázový engine je softwarová komponenta, která se stará o provádění CRUD operací s daty. Příklady u MySQL jsou InnoDB a MyISAM, kde InnoDB je výchozí engine. V základu InnoDB engine podporuje jak master-slave replikaci, tak i vysoce dostupné cluster řešení zvané InnoDB Cluster.

InnoDB Cluster [11] kombinuje tři komponenty:

- MySQL Shell - pokročilý klient a editor kódu.
- MySQL Server v kombinaci s doplňkem Group Replication dovoluje skupině MySQL instancí být vysoce dostupnými.
- MySQL Router - síťový prvek, který se stará o routing dotazů z aplikací na dostupné MySQL instance

Dále existují produkty třetích stran, jako je např. produkt společnosti Codership s názvem Galera, který je schopen dosáhnout replikace s vysokou dostupností s InnoDB. Tato práce se zaměřuje na řešení replikace u MySQL pomocí softwaru Galera.

4.2.1 Galera

Galera je plugin třetí strany pro plnou master-master replikaci s vysokou dostupností pro MySQL InnoDB engine. Pod Galerou jsou všechny uzly master, takže při výpadku jedno z nich je funkčnost nepřerušena a data zůstávají konzistentní. Řeší také konflikt při zápisu na více uzlů najednou.

Vysoká dostupnost je zaručena, pokud je mezi uzly většina, která má možnost "hlasovat" o aktuálním vedoucím clusteru, tzv. *quorum*. Quorum je zachováno, má-li cluster minimálně 3 uzly. Při výpadku jedno z uzlů je clusteru stále většina, takže je cluster schopen fungovat dále. Obecně se doporučuje mít lichý počet uzlů pro dodržení většiny.

O synchronizaci dat mezi uzly se stará tzv. *State Transfer (přenos stavu)* [12]. Existují dvě metody tohoto přenosu:

- State Snapshot Transfer (SST) - odesílá snapshot stavu celého uzlu
- Incremental State Transfer (IST) - odesílá pouze chybějící transakce místo celého stavu

Při přidaní nového uzlu do clusteru se provede zaslání dat pomocí SST.

Díky tomu, že jsou si všech uzly v Galera clustru rovny, je potřeba na zajištění vysoké dostupnosti a rozprostření zátěže směrovat dotazy na cluster inteligentním způsobem. O to se postará vyrovnávač

zátěže (load balancer). Vzhledem k tomu, že SQL dotazy jsou posílány klasicky přes TCP, stačí např. Nginx nebo HAProxy. Je ale také možné použít přímo proxy od MySQL s názvem ProxySQL, které má i podporu pro replikační protokol Galera clusteru, takže je možné vyvažovat zátěž i v rámci samotného clusteru.

Galera podporuje i funkci flow control, která aktivně omezí zápis do clustru v případě, že jeden z uzlů se zpomalí tak, že jeho data nejsou aktuální. Jakmile se daný uzel odblokuje, flow control znovu obnoví dostupnost Galera clustru.

Galera je dostupná jako plugin v různých distribucích MySQL.

- Galera Cluster (Linux a FreeBSD)
- MariaDB (je součástí základního balíčku)
- Percona XtraDB Cluster (pouze Linux)

4.3 Elasticsearch

Elasticsearch je fulltextový vyhledávací software prvně vydaný v roce 2010. Je založený na Apache Lucene knihovně, což je open-source knihovna pro vyhledávače. Je multiplatformní díky jeho implementaci v jazyce Java. Oproti MySQL se nejedná o relační databázi. Elasticsearch je bezschémová databáze, u které tedy není nutné specifikovat strukturu databáze před vložením dat. Struktura je vytvořena automaticky na základě vložených dat. [13]

Základní jednotkou dat je tzv. *Dokument*, který je ve formátu JSON. Dokumenty jsou dále sdružovány do kolekce zvané *Index*. Indexy je možné rozdělit na shards [4.1.3]. Už v základu je Elasticsearch navržen jakožto distribuovaný systém. [14]

4.3.1 Dokument

Dokument je základní jednotkou dat, která může bát indexována v Elasticsearch. Je reprezentován JSON souborem, což je formát, který je hojně využíván pro výměnu informací na celém internetu. Dokument je v ekosystému Elasticsearch něco jako je řádek v relační databázi. Dokument nemusí být pouze text, ale může být jakýmikoli daty v JSON formátu. Každý dokument má unikátní ID a datový typ, který popisuje o jaký druh dat se jedná. Dokument může představovat článek v encyklopedii, záznam z logu web serveru, technické parametry vozidla,...

4.3.2 Indexy

Index je kolekce Dokumentů [4.3.1], které mají podobné charakteristiky a jsou logicky podobné. Index je nevyšší datovou entitou, na kterou je možné se v Elasticsearch dotázat. Ve světě relačních databází je index podobný tabulce. Příklady indexu u klasického e-shopu by byl index Uživatelé, Produkty,...

4.3.3 Cluster

Už v základu se počítá s tím, že Elasticsearch bude fungovat jakožto cluster. Všechna interní logika je dělána pro optimální výkon při rozvržení zátěže na více uzlů.

4.3.3.1 Uzly [15]

Elasticsearch cluster má více rolí pro uzly podle toho, na co se specializují. Role:

- **Master** uzel je mozkem celého clustru. Rozhoduje o vytvoření nebo smazání indexu, sleduje, které uzly jsou součástí clustru, a rozhoduje o umístění shards. Pro dosažení vysoké dostupnosti je nutné mít minimálně 3 uzly typu master, aby byla dodržena většina (quorum) při výpadku jednoho z uzlů.
- **Datový** uzel drží shards, které obsahují zaindexované dokumenty. Stará se o veškeré operace s daty, takže zařizuje všechny CRUD operace, vyhledávání a agregace. Toto všechno jsou operace, které jsou velmi drahé co se týče diskového I/O, operační paměti a CPU cyklů. Datové uzly jsou ty, které je potřeba nejčastěji horizontálně škálovat.
- **Ingest** uzel slouží pro příjem a úpravu vstupních dat. Výhodné pro větší instalace, kde už je nutné tyto operace škálovat.

Uzly mezi sebou komunikují pomocí HTTP dotazů na REST API. Všechny uzly o sobě navzájem ví a jsou tak schopny přesměrovat dotaz na správný uzel.

4.3.3.2 Shards

Elasticsearch má možnost rozdělit indexy na několik menších kusů zvaných shards [viz 4.1.3]. Každý shard je sám o sobě plně funkční a nezávislý "index", který může být umístěn na jakýkoliv datový uzel v clustru. Díky této schopnosti distribuovat střípky dat Elasticsearch dokáže zajistit redundanci jak proti hardwarové poruše, tak i proti zvýšené zátěži.

4.3.3.3 Repliky

Repliky jsou kopie indexových shards. Je možné vytvořit jednu a více takových kopií. Každý dokument v indexu patří nějakému primárnímu shardu, replika kopií je jeho kopií. Repliky poskytují redundantní kopie dat pro případ hardwarové poruchy.

4.4 Kontejnerizace databázového softwaru

Pro většinu moderních a běžných databází už existuje Docker image na Docker Hub. Pro mnoho z nich jde navíc o oficiální image, takže není důvod mít obavy z narušení bezpečnosti.

Vytvoření vlastního kontejneru s instalací libovolného databázového softwaru je velmi jednoduché. Jde v podstatě o stejný proces, jako kdyby se daný databázový software instaloval v klasickém prostředí Linuxu. Rozdíly mezi klasickou instalací a tou v kontejneru nastávají, až když je kontejner spouštěn. Hlavní problémy kontejnerizovaného databázového softwaru jsou:

1. Výkon
2. Konfigurace
3. Uložiště
4. Nastavení vysoké dostupnosti

4.4.1 Výkon

Jak už bylo zmíněno v úvodu sekce Databázový software 4, virtualizace přidává určité zpoždění do procesů databázových aplikací.

4.4.2 Konfigurace

Vzhledem k tomu, že jsou kontejnery bezstavové instance a oficiální Docker images jsou dělány co nejvíce obecně, aby pokryly všechny možné aplikace daného softwaru, je potřeba aplikovat vlastní konfiguraci při každém spuštění. Je několik možností, jak tento problém vyřešit:

- Vytvořit vlastní image z oficiální image a nakopírovat vlastní konfigurační soubor přímo do hotové image.
- Použít tzv. *environmental variables*. Environmental variables jsou v podstatě pouze Linux shell (bash, sh, zsh, ...) proměnné, které se vloží do prostředí kontejneru při jeho spuštění. Aplikace v kontejneru na to musí být připravena a musí umět číst svoji konfiguraci z těchto proměnných. Jestli má aplikace svůj image na Docker Hub, tak v drtivé většině případů toto podporuje.
- Využít správce konfigurací kontejnerového orchestrátoru. Např. u Kubernetes je pro tento účel uzpůsoben objekt ConfigMap, u Docker Swarm je možné konfiguraci specifikovat v definici služby v compose souboru.

4.4.3 Uložiště

Vzhledem k tomu, že kontejnery ve výchozím stavu nemají persistentní souborový systém (viz sekce Docker Image 3.2.1.3), je potřeba pro databázový software poskytnout persistentní uložiště. U Dockeru se jedná o Volumes (viz sekce Docker Volumes 3.2.1.5).

Je nutné také myslet na to, že každá jednotlivá instance většinou má data unikátní. I když jsou instance konfigurované pro vysokou dostupnost a měly by být vzájemně zaměnitelné, toto platí pouze na venek, pro klienta připojujícího se na daný cluster. Interně mají instance většinou nějaký unikátní identifikátor. Z těchto důvodů je nutné myslet na to, aby se konkrétní instance spouštěla vždy na stejném hostu, kde je dostupný jeho svazek s daty.
Například máme-li 3 uzly (node-1, node-2, node-3) a instance jménem database-n1 běží na uzlu node-2, tak pokud bychom ji chtěli spustit na uzlu node-3 a náš svazek není přenositelný mezi uzly, tak se instance database-n1 nespustí, je nutné ji spustit na uzlu node-2.

4.4.4 Nastavení vysoké dostupnosti

Vysoká dostupnost sdružuje problémy s konfigurací a uložištěm a přidává pár dalších. Existují dva přístupy pro zprovoznění vysoké dostupnost a tím i škálování:

- Statický
- Dynamický

Oba tyto přístupy předpokládají, že kontejnerové prostředí běží pod kontejnerovým orchestrátorem. Vysoká dostupnost bez orchestrátoru je velmi náročná na manuální zásahy. Jedním z hlavních důvodů je, že kontejnery se při chybě samy nerestartují. [16]

4.4.4.1 Statický přístup

Statický přístup je jednodušší na konfiguraci, ale nejde o přístup, který by byl vhodný nasadit do produkce větší firmy. V tomto přístupu jsou všechny konfigurace, svazky a instance uzamčeny na konkrétní uzly. Pokud se počet uzlů nemění, je tato konfigurace dostačující, ale ve chvíli, kdy by se měl přidat uzel, je potřeba celou konfiguraci manuálně upravit.

4.4.4.2 Dynamický přístup

Dynamický přístup má složitou prvotní konfiguraci, ale po správném nastavení je s ním méně práce než se statickým přístupem. Vyžaduje, aby aplikace byly alespoň minimálně připraveny pro kontejnerové orchestrátory. Kontejnerizovaná aplikace musí být uzlově agnostická (nesmí jí záležet na jakém konkrétním uzlu se nachází) a totéž musí platit pro typ uložiště pro svazky. Aplikace musí být schopna si automaticky najít ostatní uzly, se kterými vytvoří cluster. Automatické hledaní uzlů má dvě možné metody:

- Dynamické DNS - místo nastavení IP adresy ostatních uzlů, spoléhá na jejich DNS jméno. Tento DNS záznam se musí automaticky obnovovat, protože kontejnery nemají ve výchozím stavu statické adresy. O obnovu DNS záznamů se stará orchestrátor.

- Vlastní síťový protokol pro *auto-discovery* (automatické objevování), který funguje tím způsobem, že prohledává síť, do které je připojen a naváže spojení s uzly, které mají spuštěn stejný protokol.

Při správném nastavení je databázový cluster schopný v reálném čase škálovat nahoru i dolů (snížení počtu uzlů) svoje uzly podle potřeby. Při přidání nového uzlu si tento uzel sám vyhledá ostatní uzly, připojí se do clustru, synchronizuje si data a do pár minut je schopen obsluhovat klienty.

Kapitola 5

Výkon databáze ve VM vs kontejneru

V této části porovnáme výkon databázového softwaru, který je spuštěn na bare-metal systému, virtuálním stroji a kontejneru. Pro zátěžové testování byl použit software Phoronix Test Suite.

Test je proveden na dvou z hlavní relačních databází, MariaDB a PostgreSQL. Obě tyto aplikace byly vybrány z důvodu, že pro ně Phoronix Test Suite poskytuje nakonfigurovaný test. MariaDB je velmi podobná MySQL, jelikož vychází ze stejného kódového základu. PostgreSQL byl vybrán jakožto kontrolní vzorek.

Testy byly prováděny na AMD sestavě s následujícími parametry uvedenými v tabulce 5.1:

Tabulka 5.1: Testovací sestava

CPUs	8 x AMD Ryzen 3 3100 4-Core Processor
RAM	Kingston 2x16GB DDR4 2666MHz ECC
HDD	Samsung 970 EVO 1TB (PCIe 3.0 NVMe)
OS	Proxmox 6.3-6 (Debian), Linux kernel 5.4.106-1

5.1 Phoronix Test Suite

Phoronix Test Suite je multiplatformní software pro spouštění automatizovaných zátěžových testů. Díky platformě OpenBenchmarking nabízí více než 450 testů. Testy se převážně skládají z XML souboru, který definuje proces celého chodu testu. Procesy nainstalovaní, přípravy testovacího prostředí, spuštění samotného testu a úklid prostředí po dokončení testu jsou nejčastěji bash skripty. Phoronix Test Suite tedy podle XML definice spustí ve správném pořadí bash skripty s vybranými parametry a poté zobrazí výsledky.
Libovolný test se přes CLI spustí tímto příkazem `phoronix-test-suite benchmark <název testu>`, veškeré nástroje jsou definované v XML testu a automaticky se stáhnou, uživatel pouze zadá parametry testu. [17]

5.2 Prostředí

Celkově byl test spuštěn ve 4 různých prostředích.

1. **Bare-metal** - systém nejblíže fyzickému hardwaru. Jedná se o operační systém Proxmox, což je virtualizační systém postavený na Debian distribuci Linuxu.

2. **Bare-metal Docker** - Docker nainstalovaný v systému Proxmox. Test v kontejneru, který je spouštěn z upravené image systému Ubuntu 20.04.

3. **Ubuntu 20.04 VM** - virtualizovaný systém Ubuntu 20.04 pomocí hypervisoru KVM.

4. **Ubuntu 20.04 VM Docker** - Docker nainstalovaný ve virtuálním stroji se systémem Ubuntu 20.04. Test je spouštěn v kontejneru, který používá stejný image jako u bare-metal Docker testu.

Každé prostředí bylo nakonfigurováno tak, aby test použil **maximálně 2 CPU jádra a maximálně 4GB operační paměti**. U virtuálních strojů to bylo zařízeno jednoduše při jejich konfiguraci v Proxmox prostředí.
U testů v bare-metal prostředí byly použity dvě metody. V bare-metal prostředí Proxmox byl příkaz spuštěn s přidanou proměnnou `NUM_CPU_CORES=2`, která podle dokumentace omezí spuštěný test na 2 CPU jádra. RAM omezena nebyla, takže v tomto prostředí test běžel se všemi 32GB RAM dostupnými, nicméně monitorováním zdrojů během testů, bylo zjištěno, že požadavky testů na RAM nikdy nepřesáhnou 4GB, takže to na výsledky testů vliv nemělo.
Docker kontejner v bare-metal prostředí byl omezen pomocí zabudovaných příkazu 5.1 v Docker CLI, které využívají cgroups [viz 3.1.1].

```
docker run --cpus 2 --memory 4GB -it localhost:5000/ubuntu_benchmark:1 /bin/bash
```

Listing 5.1: Příkaz pro spuštění Docker kontejneru pro zátěžový test

5.3 Parametry testů

5.3.1 MySQL

Tento test spočívá v nahrání ukázkové databáze o velikosti více než 2GB do instance MariaDB Serveru 10.5.2 a následného spuštění programu *mysqlslap*, který simuluje zatížení serveru. [18]

Pro tento test byl zvolen parametr 32 klientů. Test tedy simuluje zatížení databáze od 32 klientů a měří průměrný počet vyřízených dotazů za sekundu.

5.3.2 PostgreSQL

Tento test nahraje ukázkovou databázi do instance PostgreSQL serveru 13.0. Dále je možné nastavit 3 parametry.

- Násobek velikosti výchozí databáze. Výchozí databáze má 100 000 záznamů a velikost 16MB. Pro test byla vybrána hodnota 100, což znamená, že velikost databáze byla 100x větší než výchozí, což se rovná 10 000 000 záznamů s velikostí 1600MB.
- Počet klientů. Pro test byla vybrána hodnota 50, jelikož byla nejblíže hodnotě 32 klientů u MySQL testu.
- Mód. Parametr, která ovlivňuje jaké operace se při testování provádí. Tento parametr může mít jen dvě hodnoty, pouze čtení nebo čtení a zápis. Pro tento test byla vybrána hodnota čtení a zápis. Test tedy simuluje jak čtení, tak i zápis do databáze.

Výsledkem je počet transakcí za sekundu.

5.4 Výsledky

Následující tabulka 5.2 obsahuje všechny výsledky provedených testů.

Tabulka 5.2: Výsledky zátěžových testů

Prostředí	MySQL (dotazy za sekundu)	PostgreSQL (transakce za sekundu)
Bare-metal	116	1295
Bare-metal Docker	180	2513
Ubuntu 20.04 VM	55	629
Ubuntu 20.04 VM Docker	56	502

5.5 Vyhodnocení

Z výsledků je patrné, že databáze ve virtualizovaném prostředí jsou na tom výkonnostně hůře, ale dle osobních zkušeností reálný rozdíl není tak markantní, jak se jeví v těchto výsledcích. Z rozdílu mezi bare-metal a bare-metal Docker prostředím lze usoudit, že konfigurace omezení v těchto prostředích nebyla správná nebo nezafungovala jak měla. Nedává smysl, aby byla databáze o tolik výkonnější v kontejneru než mimo něj. Pro zkoumání rozdílů mezi databází v kontejneru a mimo něj bych spíše použil výsledky z virtualizovaného Ubuntu systému. Tyto výsledky jsou daleko bližší a dle mého názoru více odpovídají realitě.

Jasným závěrem je to, že virtualizace má negativní dopad na výkon databází, ale kontejnerizace má tento dopad minimální.

Kapitola 6

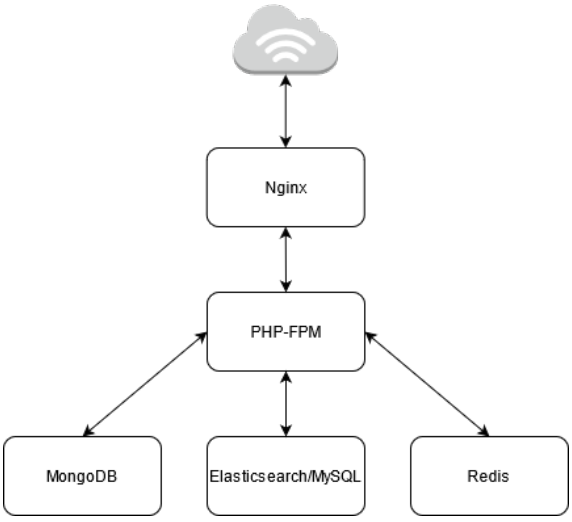
Implementace aplikace

Tato kapitola se zabývá praktickou ukázkou implementace webové aplikace s kontejnerizovanou databází Elasticsearch.

Nejprve popíšeme návrh celé aplikace a poté projdeme implementaci.

6.1 Návrh aplikace portálu Gloffer

Pro příklad webové aplikace byl vybrán portál Gloffer od společnosti Serious Investment s.r.o. Jedná se o online portál pro poptávku nových a ojetých vozů. Umožňuje zákazníkům porovnávat parametry různých vozidel a vybrat si nejvýhodnější nabídku. Diagram 6.1 zobrazuje propojení komponentů, které tvoří webovou aplikaci Gloffer.



Obrázek 6.1: Diagram návrhu Gloffer webapp

6.1.1 Komponenty aplikace

Aplikace je psána v jazyku PHP v Nette frameworku. O její sestavení se stará nástroj Composer. Pro její chod je nutný webový server s PHP podporou, Redis, MongoDB a Elasticsearch nebo MySQL databáze.

6.1.1.1 NGINX

Pro potřeby webového serveru použijeme Nginx.

Nginx je open-source webový server, který poskytuje vysoký výkon, je lehký na výpočetní zdroje a má srozumitelnou konfiguraci. Nginx je také znám svým výkonem jakožto reverzní proxy server. Díky této vlastnosti je používán v mnoha velkých firmách jakožto vstupní bod do jejich webových aplikací.

6.1.1.2 PHP-FPM 7.3

Pro implementaci jazyku PHP použijeme PHP-FPM ve verzi 7.3.

PHP je skriptovací programovací jazyk pro tvorbu webových aplikací. PHP je server-side jazyk, což znamená, že překlad a exekuce PHP kódu probíhá na straně serveru a uživateli je odeslán pouze výsledek ve formě HTML stránky. PHP je poměrně starý jazyk a za dobu své existence se pro něj vytvořilo nespočet rozšiřujících balíčků a frameworků, které práci s PHP ulehčují. FPM znamená FastCGI Process Manager. PHP-FPM je nejpopulárnější implementace PHP FastCGI. FastCGI je protokol pro interaktivní komunikaci s webovým serverem. Umožnuje paralelní zpracování několika dotazů.

6.1.1.3 MongoDB

MongoDB je v mnohém velmi podobná Elasticsearch. Jedná se také o dokumentovou databázi, která podporuje indexy. Svá data uchovává ve formátu BSON (binární JSON).

6.1.1.4 Redis

Redis je open-source NoSQL databáze, která je založena na modelu key-value. Převážně se využívá jako cachovací nástroj pro urychlení načítaní webových aplikací. Redis je převážně in-memory databáze, což znamená, že si většinu svých dat drží v operační paměti. To je jeden z hlavních důvodů, proč je tak rychlý.

6.2 Příprava prostředí

V této sekci se budeme věnovat přípravě Linux prostředí pro práci s Docker kontejnery. Cílem je vytvořit funkční Docker Swarm cluster se třemi uzly.

Pro nasazení Docker Swarm clusteru jsem si vytvořil tři virtuální stroje se systémem Ubuntu 20.04. Všechny tyto stroje jsou součástí jak interní, tak externí sítě. Interní síť bude využívat cluster ke své komunikaci. Externí síť slouží pouze pro konektivitu do internetu.

6.2.1 Instalace Dockeru

V první řadě je potřeba nainstalovat Docker na všechny virtuální stroje. Postupoval jsem podle oficiálního návodu pro Ubuntu. Tento návod zahrnuje přidání oficiálního Docker repositáře. Nejprve je potřeba přidat GPG klíč k repositáři a následně přidat samotnou adresu repositáře do zdrojů pro balíčkového manažera *apk*. Tento postup je ve výpise 6.1

```
jcech@docker-n1:~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
    gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
jcech@docker-n1:~$ echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-
    archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -
    cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Listing 6.1: Přidání Docker repositáře

Nyní už je potřeba jen obnovit cache balíčkového manažeru a nainstalovat všechny potřebné komponenty pro Docker. Postup těchto kroků je ve výpise 6.2

```
jcech@docker-n1:~$ sudo apt update
jcech@docker-n1:~$ sudo apt install install docker-ce docker-ce-cli containerd.io
```

Listing 6.2: Instalace Docker komponentů

Po dokončení instalace je dobré ještě přidat našeho Linux uživatele do skupiny Docker, aby bylo možné používat Docker CLI bez nutnosti příkazu *sudo*. Výpis 6.3 ukazuje postup. Po provedení těchto příkazů je nutné se odhlásit a zpět přihlásit, aby se změny projevily.

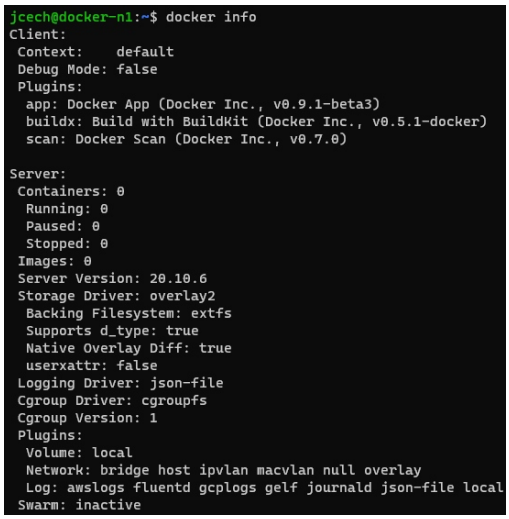
```
jcech@docker-n1:~$ sudo groupadd docker
jcech@docker-n1:~$ sudo usermod -aG docker $USER
```

Listing 6.3: Přidání uživatele do skupiny docker

Nyní bychom měli být schopni napsat `docker info` a dostat výpis informací (obrázek 6.2) o naší instalaci Dockeru. Ve výpise vidíme, ze Swarm je veden jakožto *inactive*.

6.2.2 Instalace Docker Swarm

Pro instalaci Swarmu je potřeba nejprve Swarm inicializovat na jednom z našich tří virtuálních strojů. Příkaz potřebuje jeden parametr, a to konkrétně *–advertise-addr*. Tento parametr určuje adresu, na které tento uzel bude publikovat Swarm API. V případě, že má systém pouze jedno



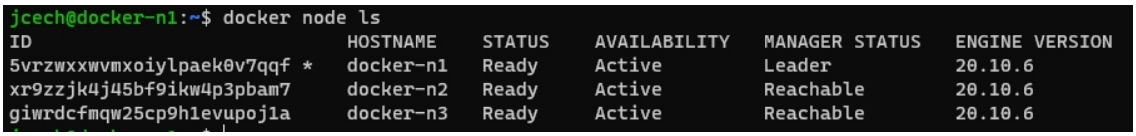
Obrázek 6.2: Výpis příkazu docker info

síťové rozhraní, není nutné tento parametr specifikovat. V našem případě ale chceme, aby Swarm API bylo dostupné pouze na interní síti.

```
jcech@docker-n1:~$ docker swarm init --advertise-addr 10.2.1.91
```

Listing 6.4: Inicializace Docker Swarm

Po úspěšné inicializaci dostaneme vygenerovaný příkaz, který obsahuje token pro přidání uzlu v roli pracovníka. Chceme ale vysoce dostupný Swarm cluster, a proto potřebujeme minimálně tři manažerské uzly.
Pomocí příkazu `docker swarm join-token manager` dostaneme stejný příkaz s jiným tokenem, který po spuštění na ostatních uzlech je přidá jakožto manažery. Jakmile jsme vygenerovaný příkaz spustili na obou zbývajících uzlech, máme hotový tří uzlový, vysoce dostupný Swarm cluster. Můžeme se o tom přesvědčit příkazem `docker node ls`. Výsledek příkazu je na obrázku 6.3



Obrázek 6.3: Výpis příkazu docker node ls

6.3 Vytvoření vlastních Docker obrazů

V této sekci vytvoříme vlastní Docker obraz pro Nginx a PHP-FPM. Vytvoření vlastního obrazu je nutné, jelikož neexistuje žádný již vytvořený, který by splňoval všechny nároky.

6.3.1 PHP-FPM

Potřebujeme nainstalovat několik různých doplňků pro PHP a některé z nich je nutné i kompilovat. Vytvoření obrazu na mém počítači trvalo přibližně 3min. Jeho výsledná velikost je 358MB. Využívám zde tzv. více etapového budovaní, což je technika, která vytvoří během budovacího procesu jeden nebo více kontejnerů, které provedou výpočetní úkol, nejčastěji jde o kompilaci programu, a následně výsledek tohoto úkolu zkopírují do finálního kontejneru. Touto metodou se ušetří místo, jelikož není nutné stahovat knihovny pro kompilovaní do finálního kontejneru a tím zvyšovat jeho velikost.
Výpis celého Dockerfile rozdělím na dvě části. První část je již zmíněný pomocný kontejner a druhá část je kontejner finální. Výpis 6.5 zobrazuje první část Dockerfile souboru pro budování vlastního obrazu.

```
ARG PHP_VER=7.3
ARG ALPINE_VER=3.12

FROM php:${PHP_VER}-fpm-alpine${ALPINE_VER} AS ext-mongodb

LABEL Maintainer="Jiří Čech" Version="1.0" \
     Description="Gloffer PHP-FPM"

ARG EXT_MONGODB_VERSION=1.8.2

#Compiling from source https://github.com/mongodb/mongo-php-driver
RUN docker-php-source extract \
    && apk update && apk add git \
    && git clone --branch ${EXT_MONGODB_VERSION} --depth 1 https://github.com/
        mongodb/mongo-php-driver.git /usr/src/php/ext/mongodb \
    && cd /usr/src/php/ext/mongodb && git submodule update --init \
    && docker-php-ext-install mongodb
```

Listing 6.5: První část Dockerfile pro vytvoření obrazu PHP-FPM

Příkazy *ARG* slouží k vytvoření proměnné v Dockerfile. Mají zde nastavené výchozí hodnoty, ale mohou být přepsány v docker-compose souboru. Zde jsem schopen pomocí nich nastavit verzi PHP, Alpine (odlehčená Linux distribuce) a verzi MongoDB plugin, kvůli kterému celý tento přídavný kontejner existuje. V příkazu *RUN* je série příkazů, které stáhnou zdrojový kód tohoto pluginu a následně nainstalují.

Na následujícím výpisu 6.6 Dockerfile pokračuje. Jedná se už o finální kontejner.

```
FROM php:${PHP_VER}-fpm-alpine${ALPINE_VER}
```

```
COPY --from=ext-mongodb /usr/local/etc/php/conf.d/docker-php-ext-mongodb.ini /usr/
    local/etc/php/conf.d/docker-php-ext-mongodb.ini
COPY --from=ext-mongodb /usr/local/lib/php/extensions/no-debug-non-zts-20180731/
    mongodb.so /usr/local/lib/php/extensions/no-debug-non-zts-20180731/mongodb.so

ARG EXT_REDIS_VERSION=5.3.2
ARG EXT_IGBINARY_VERSION=3.1.6

ARG GADMIM_UID=1000
ARG GADMIM_GID=1000
ARG GLOFFER_UID=2000
ARG GLOFFER_GID=2000

RUN addgroup -g ${GADMIM_GID} -S gadmin && \
    adduser -S gadmin -u ${GADMIM_UID} -G gadmin && \
    addgroup -g ${GLOFFER_GID} -S gloffer && \
    adduser -S gloffer -u ${GLOFFER_UID} -G gloffer && \
    addgroup gadmin gloffer

RUN apk update \
    && apk add --virtual build-dependencies \
        build-base \
        gcc \
        wget \
        git \
        icu-dev \
        gettext-dev \
        libxml2-dev \
        libxslt-dev \
        libzip-dev \
    && apk add \
        bash

RUN docker-php-source extract \
    # ext-opache
    && docker-php-ext-enable opcache \
    # ext-igbinary
    && mkdir -p /usr/src/php/ext/igbinary \
```

```
    &&  curl -fsSL https://github.com/igbinary/igbinary/archive/${
        EXT_IGBINARY_VERSION}.tar.gz | tar xvz -C /usr/src/php/ext/igbinary --strip
         1 \
    && docker-php-ext-install igbinary \
    # ext-redis
    && mkdir -p /usr/src/php/ext/redis \
    && curl -fsSL https://github.com/phpredis/phpredis/archive/${EXT_REDIS_VERSION
        }.tar.gz | tar xvz -C /usr/src/php/ext/redis --strip 1 \
    && docker-php-ext-configure redis --enable-redis-igbinary \
    && docker-php-ext-install redis \
    # other extensions
    && docker-php-ext-install bcmath calendar exif gettext intl mysqli pcntl shmop
         sockets sysvmsg sysvsem sysvshm wddx xsl zip \
    ## cleanup
    && docker-php-source delete

#Cleanup
RUN apk del build-dependencies \
    && rm -rf /var/cache/apk/*
```

Listing 6.6: Druhá část Dockerfile pro vytvoření obrazu PHP-FPM

Celý kontejner je založen za základním obrazu z oficiálního repositáře PHP na Docker Hub. Jde konkrétně o FPM variantu. První dva příkazy *COPY* slouží právě k překopírování zkompilované MongoDB knihovny z pomocného kontejneru do kontejneru finálního. Následuje znovu pár proměnných, které ovládají verzi dalších dvou PHP knihoven, a proměnné specifikující user id pro Gloffer admin účty.

První *RUN* příkaz založí již zmiňované admin uživatele a druhý *RUN* instaluje knihovny pro kompilování. V předposledním *RUN* bloku se kompilují všechny zbylé PHP knihovny. V tomto případě by bylo použití více etapového budovaní složitější, jelikož se toho zde kompiluje hodně a zkopilovaných souborů je tolik, že by bylo složité napsat všechny *COPY* příkazy z pomocných kontejnerů. Na konci se vymaže cache balíčkového manažeru, ať se ušetří co nejvíce místa a kontejner je hotov.

6.4 Elasticsearch konfigurace

Elasticsearch je konfigurován pro vysokou dostupnost a redundanci. Použil jsem přístup statického nastavení (viz 4.4.4) vysoké dostupnosti. Každý Elasticsearch uzel má pevně daný Docker Swarm uzel. Důvodem tohoto nastavení je uložiště. Jelikož využívám standardního řadiče (lokální) pro

tvoření svazků, tak nejsem schopný zaručit, že pokud by se např. jeden z Elasticsearch uzlů spustil na uzlu docker-n1, že tam bude mít svůj svazek resp. svá data. Kdybych využil například GlusterFS jakožto řadič pro svazky na Swarm clustru, tak bych měl jistotu, že by na každém uzlu jakýkoli Elasticsearch uzel našel svá data.

Nejprve je nutné pro správnou funkci Elasticsearch ještě na všech uzlech zadat tento příkaz `sudo sysctl -w vm.max_map_count=262144`. Elasticsearch se bez tohoto nastavení kernelu nespustí.

Samotný konfigurační soubor Elasticsearch je na výpisu 6.7. Jedná se pouze o konfigurační soubor pro uzel 1, ale ostatní se liší pouze v řádku `discovery.seed_hosts`, kde se uvádí DNS jméno ostatních členů clustru. Zakomentované příkazy dole slouží k zapnutí HTTPS protokolu pro komunikaci mezi jednotlivými uzly a také komunikace s API. Pro nastavení HTTPS je potřeba vygenerovat certifikáty a umístit je na dané lokace v kontejnerech. Za zmínku ještě stojí *network.host*, který je nastavený na poslouchání na všech síťových adaptérech kontejneru. Což může být nežádoucí, když je kontejner součástí více sítí.

```
#bootstrap.memory_lock: true
discovery.seed_hosts: node2, node3
cluster.initial_master_nodes: node1, node2, node3
discovery.zen.ping_timeout: 5s
discovery.zen.commit_timeout: 5s
node.name: node1
cluster.name: "gloffer"
network.host: 0.0.0.0
xpack.security.enabled: false
#xpack.security.http.ssl.enabled: true
#xpack.security.http.ssl.keystore.path: /usr/share/elasticsearch/config/
    certificates/${node.name}/${node.name}.p12
#xpack.security.http.ssl.truststore.path: /usr/share/elasticsearch/config/
    certificates/${node.name}/${node.name}.p12
#xpack.security.http.ssl.key: /usr/share/elasticsearch/config/certificates/${node.
    name}/${node.name}.key
#xpack.security.http.ssl.certificate: /usr/share/elasticsearch/config/certificates
    /${node.name}/${node.name}.crt
#xpack.security.http.ssl.certificate_authorities: /usr/share/elasticsearch/config/
    certificates/ca/ca.crt
#xpack.security.http.ssl.verification_mode: certificate
#xpack.security.transport.ssl.enabled: true
#xpack.security.transport.ssl.verification_mode: certificate
#xpack.security.transport.ssl.keystore.path: /usr/share/elasticsearch/config/
    certificates/${node.name}/${node.name}.p12
```

```
#xpack.security.transport.ssl.truststore.path: /usr/share/elasticsearch/config/
    certificates/${node.name}/${node.name}.p12
#xpack.security.transport.ssl.certificate: /usr/share/elasticsearch/config/
    certificates/${node.name}/${node.name}.crt
#xpack.security.transport.ssl.key: /usr/share/elasticsearch/config/certificates/${
    node.name}/${node.name}.key
#xpack.security.transport.ssl.certificate_authorities : /usr/share/elasticsearch/
    config/certificates/ca/ca.crt
```

Listing 6.7: Konfigurační soubor Elasticsearch clustru

6.5 Nasazení aplikace

Pro nasazení celé aplikace na Docker Swarm je každá z komponent definována jako služba [viz 3.3.1] v docker-compose souboru. Jsou zde definovány konfigurační soubory, svazky a sítě.

Na prvním výpisu 6.8 lze vidět definici služeb *nginx* a *php-fpm*. Položka *build* je u obou z těchto služeb důležitá, jelikož umožnuje před spuštěním celé aplikace postavit tyto dva upravené obrazy. *context* odkazuje na složku relativní k *docker-compose.yml* souboru, kde se nachází daný Dockerfile jakožto definice vlastního Docker obrazu. Pomocí položky *args* je možné měnit proměnné v Dockerfile souborech.

```
version: "3.7"

services:

  nginx:
    build:
      context: ./nginx
      args:
        NGINX_VER: 1.19.2
    image: jcech_nginx:latest
    networks:
      - esnet
    ports:
      - target: 8080
        published: 80
        protocol: tcp
        mode: ingress
      - target: 8081
```

```
      published: 8081
      protocol: tcp
      mode: ingress
  configs:
    - source: nginx
      target: /etc/nginx/conf.d/gloffer.conf
  deploy:
    mode: replicated
    replicas: 1
    restart_policy:
      condition: on-failure
      delay: 10s
      max_attempts: 3

php-fpm:
  build:
    context: ./php-fpm
    args:
      PHP_VER: 7.3
      ALPINE_VER: 3.12
  image: jcech_php-fpm:latest
  networks:
    - esnet
  ports:
    - 9000:9000
  deploy:
    mode: replicated
    replicas: 1
    restart_policy:
      condition: on-failure
      delay: 10s
      max_attempts: 3
```

Listing 6.8: Docker-compose 1

Následují definice MongoDB a Redis, které jsou víceméně ve výchozím stavu. Dále je zde definován jeden z uzlů Elasticsearch.
Důležité nastavení je u něj *deploy.placement.contrainsts*, které zaručuje, že tento konkrétní uzel se spustí pouze na Swarm uzlu se jménem docker-n1. Dalším duležitým nastavením je *memlock* u

parametru *ulimits*. Není-li memlock nastaven na hodnoty -1, tak se Elasticsearch může odmítnout spustit. Některé systémy nemusí podporovat nastavení *memlock* u kontejneru, pak je nutné odkomentovat první řádek v konfiguračním souboru Elasticsearch, který si memory locking vynutí jiným způsobem. Elasticsearch uzlu se zde také přiřazuje svazek a dva konfigurační soubory, jeden z nich je ve výpisu 6.7. Je zde také definován *healthcheck*, který periodicky zkouší příkaz v položce *test* a podle výsledku je daný kontejner vyhodnocen jako zdravý nebo ne.

```
  node1:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.9.1
    healthcheck:
      test: curl -s https://localhost:9200 >/dev/null; if [[ $$? == 52 ]]; then
          echo 0; else echo 1; fi
      interval: 30s
      timeout: 5s
      retries: 3
      start_period: 45s
    configs:
      - source: es-node1
        target: /usr/share/elasticsearch/config/elasticsearch.yml
      - source: jvm-options-data
        target: /usr/share/elasticsearch/config/jvm.options
    networks:
      - esnet
    ports:
      - target: 9200
        published: 9200
        protocol: tcp
        mode: host
    volumes:
      - esdata1:/usr/share/elasticsearch/data
    ulimits:
      memlock:
        soft: -1
        hard: -1
    deploy:
      placement:
        constraints: [ node.hostname == docker-n1 ]
      endpoint_mode: dnsrr
      mode: replicated
```

```
      replicas: 1
      resources:
        limits:
          memory: 4G
```

Listing 6.9: Docker-compose 2

Poslední definice služby je *Kibana*, což je vizualizační nástroj pro Elasticsearch. Na konci souboru už jsou jen deklarace jedné hlavní sítě *esnet*, deklarace čtyř svazků a definice všech použitých konfiguračních souborů.

```
  kibana:
    image: docker.elastic.co/kibana/kibana:7.9.1
    networks:
      - esnet
    #  - proxy
    ports:
      - 5601:5601
    configs:
      - source: es-kibana
        target: /usr/share/kibana/config/kibana.yml
    deploy:
      placement:
        constraints: [ node.hostname == docker-n1 ]
      mode: replicated
      replicas: 1
      update_config:
        failure_action: rollback
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
        delay: 10s
        max_attempts: 3


networks:
  esnet:
    driver: overlay
    name: esnet
```

```
volumes:
  esdata1:
  esdata2:
  esdata3:
  mongo-data:

configs:
  es-node1:
    name: es-node1
    file: elasticsearch/es-node1.yml
  es-node2:
    name: es-node2
    file: elasticsearch/es-node2.yml
  es-node3:
    name: es-node3
    file: elasticsearch/es-node3.yml

  es-kibana:
    name: es-kibana
    file: elasticsearch/es-kibana.yml

  nginx:
    name: nginx
    file: nginx/config/gloffer.conf

  jvm-options-data:
    name: jvm-options-data
    file: elasticsearch/jvm.data.options
```

Listing 6.10: Docker-compose 3

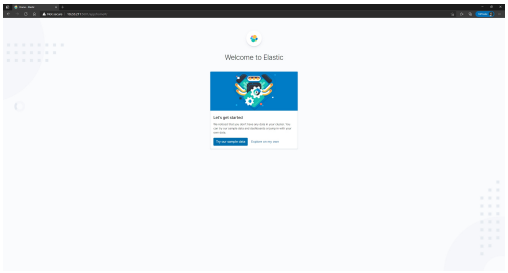
Samotné nasazeni aplikace jsou pouhé dva příkazy. Je nutné, abychom se nacházeli v kořenu složky, kde je soubor *docker-compose.yml*. První příkaz `docker-compose build` spustí výstavbu dvou vlastních obrazů. Rychlost provedení závisí na rychlosti počítače, kde běží Swarm. Poslední příkaz je `docker stack deploy -c docker-compose.yml jcech_gloffer_app`, poslední parametr je název a je libovolný. Nyní si můžeme ověřit, že všechny služby se v pořádku spustily pomocí příkazu `docker service ls`. Na výpise 6.4 je vidět výsledek.

```
jcech@docker-n1:~/Gloffer$ docker service ls
ID             NAME                        MODE         REPLICAS   IMAGE                                          PORTS
ic079vin6vgv   jcech_gloffer_app_kibana    replicated   1/1        docker.elastic.co/kibana/kibana:7.9.1          *:5601->5601/tcp
cqpr3i5mqr7l   jcech_gloffer_app_mongodb   replicated   1/1        mongo:latest                                   *:27017->27017/tcp
i7gghokixp6d   jcech_gloffer_app_nginx     replicated   1/1        jcech_nginx:latest
pohgwuwbcl9n   jcech_gloffer_app_node1     replicated   1/1        docker.elastic.co/elasticsearch/elasticsearch:7.9.1
lvmflxkn19al   jcech_gloffer_app_node2     replicated   1/1        docker.elastic.co/elasticsearch/elasticsearch:7.9.1
x1r75sey5p2v   jcech_gloffer_app_node3     replicated   1/1        docker.elastic.co/elasticsearch/elasticsearch:7.9.1
o75h6ha44y0f   jcech_gloffer_app_php-fpm   replicated   1/1        jcech_php-fpm:latest                           *:9000->9000/tcp
uowguq6x5vyp   jcech_gloffer_app_redis     replicated   1/1        redis:latest
```

Obrázek 6.4: Výpis příkazu docker service ls

Funkčnost Kibany a webového serveru si můžeme ověřit přes webový prohlížeč. Známe-li veřejnou IP jakéhokoli uzlu ze Swarmu, tak bychom se po zadání portu měli na služby dostat. Obrázky 6.5 a 6.6 ukazují funkční web i portál Kibana.

Obrázek 6.5: Gloffer úvodní stránka



Obrázek 6.6: Funkční Kibana

6.6 Škálování aplikace

Samotné škálování v Docker Swarm je záležitost pouze jednoho příkazu. Je ale nutné myslet na to, že kontejner přidaný škálováním se může spustit na jakémkoli uzlu ve Swarm clustru, nemá-li v definici své služby uvedenou žádnou restrikci pro nasazení [viz definice Elasticsearch služby 6.9]. Jeho obraz se tedy musí nacházet na všech uzlech Swarm clustru. To není problém u kontejnerů, jejichž obraz je volně dostupný z Docker Hub, protože při nedostupnosti obrazu na uzlu si jen Swarm jednoduše stáhne přes internet. Problém nastává u našich upravených obrazů Nginx a PHP-FPM, které existují pouze lokálně na uzlu, kde byly postaveny. Jedním z řešení je obraz postavit na všech zbylých uzlech. Jednoduše přenese složku *nginx* na zbývající uzly a v této složce na každém z uzlů spustím příkaz `docker build -t jcech_nginx:latest .`, tečka na konci příkazu říká procesu, že má obraz postavit z aktuální složky, parametr *-t* nastaví název obrazu.

Máme-li nyní upravený Nginx obraz na všech uzlech, tak pomocí příkazu `docker service scale jcech_gloffer_app_nginx=2` jej naškálujeme na dva běžící kontejnery. Zobrazení webové stránky nyní funguje tak, že po zadání libovolné veřejné IP adresy jednoho ze tří hostů, se dotaz interně přesměruje na hosta, který má aktuálně běžící nginx kontejner a nasměrujeme dotaz na něj. Zároveň při případu, kdy by přicházelo na Nginx server více dotazů, tak je Swarm rovnoměrně rozdělí mezi aktuálně běžící kontejnery a tím rozdělí zátěž. [19]

6.6.1 Škálování Elasticsearch služby

Díky použití statické konfigurace [6.4] a závislosti na uložišti, není škálování Elasticsearch služby tak jednoduché, jako tomu bylo u Nginx služby. Je nutné provést tyto změny, abychom mohli přidat pouze jeden nový uzel do Elasticsearch clustru.

- Vytvořit novou konfiguraci pro samotnou aplikaci [6.7]
- Upravit konfigurace stávajících Elasticsearch aplikací, aby reflektovaly přidání nového uzlu. Jde o úpravu řádku 2 a 3 v tomto [6.7] konfiguračním souboru.
- Přidat deklaraci nového uložiště a nově vytvořeného konfiguračního souboru do souboru *docker-compose.yml* [6.10]
- Vytvořit novou definici služby v souboru *docker-compose.yml* [6.9]

6.7 Demonstrace vysoké dostupnosti

V této části nasimulujeme výpadek jednoho celého uzlu Swarmu, ověříme funkčnost Elasticsearch clustru, uzel zpátky oživíme a zkontrolujeme automatické uzdravení clustru.

6.7.1 Výchozí stav

Nejprve zkontrolujeme, že clustr běží v pořádku. Nejjednodušší způsob je poslat dotaz na Elasticsearch API. Je k tomu potřeba Linuxový nástroj *curl*. Z terminálu jednoho z Docker Swarm uzlů pošleme tento příkaz.

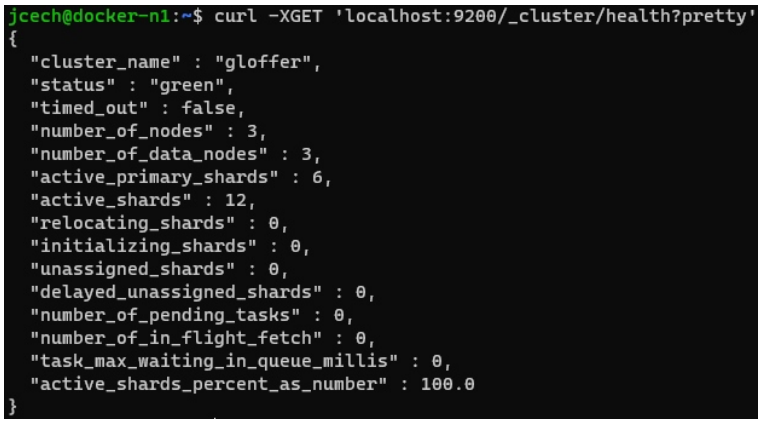
```
curl -XGET 'localhost:9200/_cluster/health?pretty'
```

Listing 6.11: Dotaz na Elasticsearch API

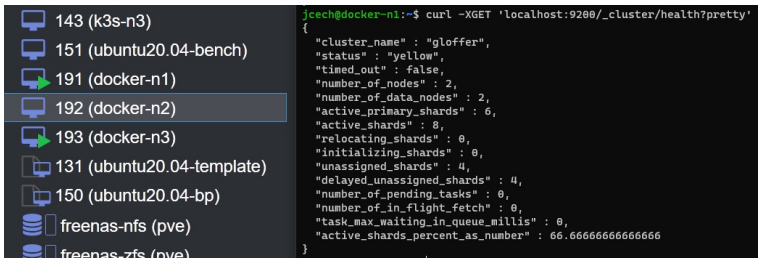
Výsledkem by měl být výpis aktuálního zdraví Elasticsearch clustru. Tento výpis je na obrázku 6.7

6.7.2 Vypnutí jednoho z uzlů

V další fázi simulujeme výpadek jedno ze tří virtuálních strojů, na kterých běží Docker Swarm cluster. Vypnutí je realizováno přes webové rozhraní virtualizačního systému Proxmox. Hned po vypnutí pošleme dotaz na API 6.11 znovu a tentokrát bychom měli vidět jinou odpověď. Vypnutí uzlu a následná odpověď Elasticsearch API je na obrázku 6.8 Na obrázku 6.8 je vidět vypnutý uzel *docker-n2* a stále funkční uzly n1 a n3. Odpověď API už ukazuje pouze 2 uzly v počtu uzlů. Důležitá je položka *status*, která měla hodnotu *green*, když fungovaly všechny 3 uzly a nyní má hodnotu *yellow*. Hodnota *yellow* je podle dokumentace [20] stav, kdy je cluster funkční, ale výpadek dalšího uzlu by už znamenal jeho nefunkčnost.



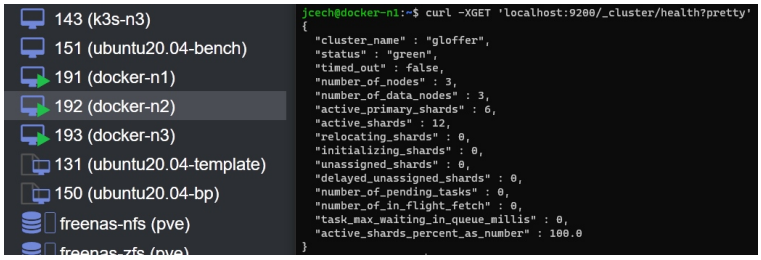
Obrázek 6.7: API odpověď zdravého Elasticsearch clusteru



Obrázek 6.8: Vypnutí uzlu Docker Swarm a odpověď Elasticsearch API

6.7.3 Návrat do plně funkčního stavu

Díky Docker Swarm orchestrátoru a vlastností Elasticsearch by vracení se do plně funkčního stavu mělo být automatické. Po znovu zapnutí uzlu n2 by měl Swarm poznat, že je uzel znovu funkční a obnovit kontejnery, které by na něm měly běžet. Obrázek 6.9 ukazuje stav po znovu zapnutí virtuálního stroje.



Obrázek 6.9: Zapnutí dříve vypnutého uzlu Docker Swarm a odpověď Elasticsearch API

Proces automatické obnovy je také možné si ověřit příkazem `docker service ls`, jehož výstup by měl být totožný s výsledkem po prvotním nasazení celé aplikace [6.4].

Ověřili jsme si, že u Elasticsearch i Docker Swarm vysoká dostupnost a redundance fungují, jak by měly. I s jedním chybějícím uzlem byly obě aplikace schopny fungovat jako před výpadkem uzlu

a dostupnost celé aplikace nebyla nijak ovlivněna.

Kapitola 7

Závěr

Cílem práce bylo prozkoumání metodik vytváření a používání kontejnerizovaných databázových serverů s důrazem na jejich škálování a vysokou dostupnost. Začátek práce prošel teoretické základy pro pochopení, z čeho principy kontejnerizace vychází a jak se vyvíjely.

Dále se pokračovalo hlouběji do problematiky kontejnerizace a začaly se objevovat první otázky ohledně možných problémů s výkonem kvůli virtualizaci. To bylo ale vyváženo sekcí s replikačními technologiemi, které u obou zkoumaných databází působily kvalitně a funkčně.

Po prozkoumání obou dvou různých druhů databázových technologii, MySQL a Elasticsearch, a jejich systémů ukládání dat, bylo zřejmé, že Elasticsearch je jasnou volbou pro databázi, která má fungovat v kontejnerovém ekosystému. Relační model má bezpochyby v určitých oblastech výhodu oproti dokumentovému modelu Elasticsearch (např. v užší kontrole nad daty), ale co se týče kontejnerů, tak MySQL model není ideální.

Hned ve čtvrté kapitole po výčtu vlastností Elasticsearch je ale zřejmé, že se implementace od teorie liší. Nutnost řešit speciální druhy uložišť, aby byla zajištěna vysoká dostupnost, působí už trošku složitěji.

Zátěžové testy v kapitole pět nebyly provedeny takovým způsobem, aby bylo v otázce ovlivnění výkonu virtualizací úplně jasno, protože rozdíly mezi nevirtualizovaným a virtualizovaným prostředím byly až moc velké. To znamená, že je zde mnoho neznámých, které mohou výkon ovlivnit. Každopádně pomohly alespoň ke směrodatné odpovědi.

Při implementaci aplikace se ukázaly všechny dřívější poznatky jako pravdivé. Nastavení bylo složité a jen malá chyba znamenala, že celá aplikace nefungovala. Ale když se vše povedlo nastavit správně, tak správa a obsluha aplikace byla velmi jednoduchá. Mohl jsem si dovolit celou aplikaci smazat a zpět nahodit během dvou minut.

Abych odpověděl na první otázku z úvodu, kontejnerizace je výhodou pro databáze a jejich správce. To platí ale pouze, jsou-li ochotni investovat čas a peníze do správné a robustní konfigurace. S polovičně funkční konfigurací správce s databází téměř jistě za určitý čas narazí. Investice do

kvalitní konfigurace se nepochybně vyplatí, například zachytí-li kontejnerový orchestrátor výpadek systému a obnoví ho, může tak firmě ušetřit milióny.

Literatura

1. *What is virtualization?* [Online]. [cit. 2021-4-27]. Dostupné také z: https://opensource.com/resources/virtualization.
2. RESELLERCLUB. *Type 1 and Type 2 Hypervisors* [online]. [cit. 2021-4-28]. Dostupné také z: https://medium.com/teamresellerclub/type-1-and-type-2-hypervisors-what-makes-them-different-6a1755d6ae2c.
3. *Docker overview* [online]. [cit. 2021-4-28]. Dostupné také z: https://docs.docker.com/get-started/overview/.
4. *What is Containerization?* [Online]. [cit. 2021-4-28]. Dostupné také z: https://www.citrix.com/solutions/app-delivery-and-security/what-is-containerization.html.
5. *namespaces(7)* [online]. [cit. 2021-4-28]. Dostupné také z: https://man7.org/linux/man-pages/man7/namespaces.7.html.
6. *How nodes work* [online]. [cit. 2021-4-28]. Dostupné také z: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/.
7. PACHEV, Sasha. *Understanding MySQL Internals* [online]. O'Reilly Media, Inc., [cit. 2021-4-29]. Dostupné také z: https://www.oreilly.com/library/view/understanding-mysql-internals/0596009577/ch01.html.
8. *Redis* [online]. [cit. 2021-4-28]. Dostupné také z: https://redis.io/.
9. DORMANDO. *Memcached* [online]. [cit. 2021-4-28]. Dostupné také z: https://memcached.org/.
10. *Achieving Ultra-Low Latency in Trading Infrastructure* [online]. [cit. 2021-4-28]. Dostupné také z: https://www.exegy.com/2020/02/ultra-low-latency-trading-infrastructure/.
11. *InnoDB Cluster* [online]. [cit. 2021-4-29]. Dostupné také z: https://dev.mysql.com/doc/refman/8.0/en/mysql-innodb-cluster-introduction.html.
12. *State Transfers with Galera Cluster* [online]. [cit. 2021-4-29]. Dostupné také z: https://galeracluster.com/library/documentation/state-transfer.html.
13. GORMLEY, Clinton; TONG, Zachary. *Elasticsearch: the definitive guide.* [B.r.]. ISBN 1449358543.

14. ABUEG, Ralf. *What is Elasticsearch and what is it used for* [online]. [cit. 2021-4-28]. Dostupné také z: https://www.knowi.com/blog/what-is-elastic-search/.

15. *Node: Elasticsearch Guide 7.12* [online]. [cit. 2021-4-28]. Dostupné také z: https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-node.html.

16. SHIVAKUMAR, Shailesh Kumar. *Architecting high performing, scalable and available enterprise web applications*. [B.r.]. ISBN 9780128022580.

17. *Phoronix* [online]. [cit. 2021-4-28]. Dostupné také z: https://www.phoronix-test-suite.com/.

18. *mysqlslap* [online]. [cit. 2021-4-29]. Dostupné také z: https://mariadb.com/kb/en/mysqlslap/.

19. *Use swarm mode routing mesh* [online]. [cit. 2021-4-30]. Dostupné také z: https://docs.docker.com/engine/swarm/ingress/.

20. *Cluster health API* [online]. [cit. 2021-4-30]. Dostupné také z: https://www.elastic.co/guide/en/elasticsearch/reference/current/cluster-health.html.

Příloha A

Obsah

- App
  - elasticsearch - konfigurační soubory
  - nginx - Dockerfile a konfigurační soubory
  - php-fpm - Dockerfile
  - *docker-compose.yml*
- Stresstest - výsledky zátěžového testu
  - htop - snímky obrazovky z programu *htop*
  - results - snímky obrazovky jednotlivých výsledků testů
  - *PhoronixTest_Dockerfile* - Docker obraz použitý v zátěžových testech